Coding Workshop

Learning to Program with an Arduino

Lecture Notes

Table of Contents

How?

- Programming doesn't have to be scary, in fact much easier than French or Spanish!
- With just a few keywords, should be able to write many simple programs
- Here's an example of a simple program:
  `print("Hello, World!")`
- It consists of a *function*, `print( )`, and a *value*, `"Hello, World!"`; we will come back to these later
- It is written in a popular and fairly easy language, python; We encourage you to learn this if you gain an interest in coding!

How?

- The Arduino uses a slightly stricter language, a dialect of C++
- The patterns you learn here will in fact work in many languages ('C-like' languages, which unfortunately does not include python)
- Don't worry if you don't take everything in first time round! We have provided a cheat sheet, and there are some exercises for you to try which should help you put these concepts into practice
- Hopefully you will come to realise that writing programs is just breaking down tasks into simple instructions so that a stupid computer can repeat them; In reality you do not need to know much to write any program, as computers do not understand many different instructions, and only a very few are actually necessary

C++ Values

- Values are items of data, and in C++ it is quite important to explicitly distinguish between different types of data
- Most important are numbers; there are two main types:
  - Whole numbers, known as `int`; e.g. `3`, `-4`
  - Decimal numbers, known as `float`; e.g. `3.0`, `2.71`
- We will also need truth values, known as `bool`, i.e. `true` and `false`
- *In an earlier example you also met textual data; we shouldn't need it here but we refer to single letters/numbers/etc as* `char`*, e.g.* `'%'`*, and blocks of text as* `char[]`*, e.g.* `"Hello, World!"`

Assignment

- Believe it or not, you're almost ready to program!
- Firstly, it will be nice to be able to remember values; we do this through *assignment*:
  - Let's analyse a simple example, `int x = 7;`
  - First note that we had to *declare* what type of value `x` is; Computers are a bit stupid, and so sometimes we have to help them along a bit
  - This is also why we include a *semicolon*; Though languages can differ in the way they're written, most have punctuation to indicate the end of an instruction or *statement*
  - We can now use `x` to refer to `7` whenever we like!
  - We can also change `x` at any time, in this way the `=` sign differs from maths– `x = 42;` –note that we have now *changed* the value of `x`, also note that the computer has already been told that `x` is a whole number, and it doesn't like to be told twice!

Manipulating Values

- Computers excel at arithmetic and decisions, and almost all computation is simple combinations of these
- Arithmetic is achieved as simply as using the following four symbols: `+`, `-`, `*`, `/`, with parentheses where appropriate!
- Computers are quite strict, and dividing whole numbers will give you the quotient, you can get the remainder with `%`
- Continuing on with our example, let's try to find the product of 7 and 6:

```
int x = 7;
int y = 6;
x = x * y;
```

- You may wonder whether the last line makes sense... In algebra this would mean $y = 1$ or $x = 0$, but here the `=` sign means *assignment*, so the computer first works out the answer on the right, and then assigns it to the name on the left

Conditions and Control

- The last thing we need to be able to write any program is the ability to make decisions
- We will consider two decision making *structures* or *blocks*: `if` and `for` (there are others but they don't really do anything else)
- `if` lets you decide whether to run a *block* of code (set of statements) based on a previous result, for example if `x` is even; it is often accompanied by `else` which runs a different block otherwise
- `for` lets you run a block of code multiple times, 'for every time that something is true', though it is easier to think of as doing something for every number in a given range

How to Decide?

equality	`==`	`x == 4`	is `x` 4?
inequality	`< >`	`x > 4`	is `x` more than 4?
	`<= >=`	`x <= 4`	is `x` less than or equal to 4?
not	`!`	`!( x > 4)`	is `x` *not* greater than 4?
	`!=`	`x != 4`	is `x` *not* 4?

- You can replace `x` above with any *expression*, for example to test for evenness: `(x % 2) == 0` tests if the remainder of dividing `x` by `2` is 0
- You can also save the truth of these tests to a `bool` *variable*: `bool is_x_even = ((x % 2) == 0);`

`if` Blocks

```
if (x == 42) {
    float y = 3.14159;
}
```

- The above shows the structure of an `if` block
- Try to identify the parts discussed earlier; notice the condition and the subsequent code block–code blocks are punctuated by curly braces
- We can now try to put together what we've learned and include the `else` structure

```
int x = 37;
if ((x % 2) == 0) {
    x = x / 2;
} else {
    x = 3*x + 1;
}
```

`for` Blocks

```
for (int i = 0; i < 5; i = i + 1) {
    x = x + i;
}
```

- The `for` block is a little more complicated, let's try to understand it:
  - Like in the `if` block, we have some parentheses and a code block; the parentheses are more complicated though and contain exactly three statements
  - The result is that we add the numbers 0, 1, 2, 3, 4 to `x` one-by-one by *iterating* over the values $0 \leq i < 5$
  - The first says that we want a whole number `i` starting at 0; the second says we should run the code block as long as $i < 5$; the third says that after each step we should increase `i` by 1
- You may see a common abbreviation for the third expression, `i++`, to *increment* `i`; and similarly, `i--` to *decrement* it

Congratulations!

- Though it may not seem like it, you can now write any computer program imaginable!
- There are some more tricks to be learned to write efficient and concise code, and we will expose you to one of them in the next slide (though you need not make use of it yourself today)
- To test your understanding, try to identify the purpose of the following code:

```
int x = 1;
int n = 10;
for (int i = 1; i <= n; i++) {
    x = x * i;
}
```

Functions

- In large projects it can become repetitive writing code that does the same thing over and over again, for example calculating powers of numbers
- For convenience, most programming languages let you define *functions* or *subroutines*; these are (almost) just like mathematical functions, and in fact functions like `sin` do exist!
- These also let you do special things such as display output (`printf`) to the screen so you don't need to understand hidden complexities like *interrupts*; unfortunately the arduino doesn't have a screen, but we will be using other output functions to control some LEDs
- You have seen how functions are called in our first example, you state their name and give them *parameters*:
  `float x = sin(1.5708);`

Arduino Functions

- In C++ all code must actually go in a function; for the Arduino there are two main functions - `setup` and `loop`
  - When the Arduino is turned on, it *calls* `setup` , running any code within, for example to setup variables and electrical pin modes
  - After setup, the Arduino then calls `loop` repeatedly until it loses power
- *Note, you can also add comments to your code using* `//`

```
int x;
void setup() {
    x = 42;
}
void loop() {
    x--; // decrement x
}
```

Arduino Functions

- There are around 9 very good Arduino functions to know, but for today it is good enough for us to know three of them:
  - `pinMode`, measure (`INPUT`) or set (`OUTPUT`) the voltage
  - `digitalWrite`, set the voltage to 5 V (`HIGH`) or 0 V (`LOW`)
  - `delay`, stop everything for a given number of milliseconds
- Here's an example using these to blink an LED attached to pin 13 (and a 1 kΩ resistor!) on and off:

```
int pin = 13;
void setup() { pinMode(pin, OUTPUT); }
void loop() {
  digitalWrite(pin, HIGH); delay(1000);
  digitalWrite(pin, LOW);  delay(1000);
}
```